Plagirism Detection Software Report

Austin Agbo

Information Systems MSc

Cloud Computing

*Abstract*—This document is a description of Plagiarisms Detection Software designed around 2 public clouds (GAE & AWS) & a hybrid cloud (Open Stack). It discusses the parts of the cloud business as a whole, implementation and testing of the system. Also does a scale analysis for more instances added to test performance and timing. Lastly carries out a critique of the approach chosen to build such a system.

Keywords—cloud; aws; server; openstack; (key words)

# Introduction

Cloud Services – The response to heavy lifting, paying others to carry the heavy luggage you are unfit to carry, easy processing of large data, this implies paying the bell boy for any more of his time you utilize. Having this in mind, a customer should manage available resources in a cost effective and most flexible manner possible.

For the context of this study, the system that will be analyzed is plagiarism detection software. A system that compares Document A – Document B i.e. Source document to suspicious document. Expected result is the common found in the suspicious. How much exists and where is it?

To build this, key understanding of web development processes and different type of cloud interaction is required – being aware of a fixed amount that is expected for the system at hand and hence manipulations based on this fact. In times of progression to the Internet of things - virtualization, ubiquity, prevalence of high-speed networking is paramount. This drives a change from an institution to deploy and manage physically own servers. Essentially, economically and performance wise, a cloud system might be better. Though management and maintenance prompts a re-mergence of timesharing, usage-based pricing and shared resources, the outcome is substantial [1].

“The best way to eat an elephant is to share it into lesser chunks” – This system handles the data involved by having two sections, which are the Indexing and Matching. The logical two steps - used to identify repetition. In these two steps, mapping and reducing occurs: where certain keys are mapped to certain values and then reduced to have more concise data for identification.

This map-reduce method is provided by; mapper and reducer interfaces. Our Application has to implement them and therefore this forms the core of the job. The software framework used here is Apache Hadoop, which provides support for distributed shuffles. Its file structure is known as Hadoop Distributed File Structure (HDFS). The system would have master/slave architecture, were a cluster consists of a single NameNode. This master server is where the map-reduce aspect of the plagiarism application will live. The back end of the application, it manages the file system and regulates access to files by clients. In addition there will be a specified number of Data nodes, usually one per node in the cluster, which manages storage attached to the nodes that they can run on.

Application takes in an input of a series of text files, where input specification for the jobs is ‘input-split’ that represents the data (text files) to be processed by an individual mapper. Default ‘input-split’ is ‘file-split’ which sets ‘map.input.file’ to the path of the input file for the logical split.

The output format describes the output-specification for a Map-Reduce job; Text is the default Output Format. The key itself is the representation of the document while the paths of the documents (directories) are the values.

The Cloud System as a whole is a Software as a Service that uses a platform as a service and an Infrastructure as a Service i.e. two public clouds (Google App Engine & Elastic Map Reduce) plus a private cloud (Open Stack). Requests can be made to the first two via http and third party interfaces respectively. Whereas the latter only allows requests to an instance of images on open stack by the administrator via Secure Shell (when not in premise of the open stack), Therefore it can be described as a hybrid.

The EMR was chosen for the implementation and not EC2 due to it being easier to configure and set instructions to; using the Boto Interface, which is also written in python as the User Interface (Front End) that is being deployed on Google App Engine. The EC2 would have taken as much effort to install and set hadoop on as it took setting up Open Stack. On top of this, cost is certain with EMR where cost is for the hour and being precise in terms of the existing interface that allows monitoring of the application, its log and other easily attachable features.

To produce the index, a window size is considered where ever next window is a word away. All the patterns that can be gotten from a window are used to represent that window as the key of the map. The values are the document, start and end positions of that pattern. This optimizes the communication cost and is essential for a good Map-Reduce Algorithm.

# Implementation & Testing of System

## GAE

The engine hosts the frontend/view for the application. With a html index page that sits on a python application, it allows a user to enter values through a form that it posts to an alternative page as a string in the same template of that index page. The values derived from the user starts the EMR/Open Stack based on the amount of nodes and the type of workflow set by the user. Time at point of submit button is recorded and posted to the alternative page.

## S3

A bucket named agbocloudmsc2015mapreduce was created which contains four folders: a rough for storing source and suspicious document, a folder for the Index Map – Index Reduce programs, a folder for the Match Map – Match Reduce programs, a folder for indexed results, and a folder for matched results. As stated in the GAE, these folders are used to point to where data should be picked up from and dropped by the EMR and Open Stack.

## EMR

All settings and configuration for the EMR are done using the boto interface written in a python application that lives on the instance on open stack. It is ignited by the submit button depending on the workflow chosen. Minding all Access, private keys and other settings to start up the cluster are set in this python application.

## Open Stack

While being on private premise, an image was made of a Linux physical machine. An instance was created, which could be accessed via secure shell by the administrator anywhere else. Installed on this instance server is the latest Hadoop version 2.7.0. A python script that polls and awaits a url generated from the front end application is being retrieved. It gets only a new request made which is apparent by the time recorded when submit was hit. It appends this value to a text file; it won’t record the same values of string with the same time value.

This script gepage.py, it’s job is to await a new request from the front end and then runs either an EMR/Open Stack Hadoop Job with the string values in context to start the process of indexing or matching depending on the values it receives.

Note, it points to different folders depending on the workflow it is to use. The S3 sync is used from the AWS CLI to download files to the open stack instance (master-node) this provides the text files to where they would be used to start the hadoop process if the user requires the workflow to be initiated by Open Stack as the case may be.

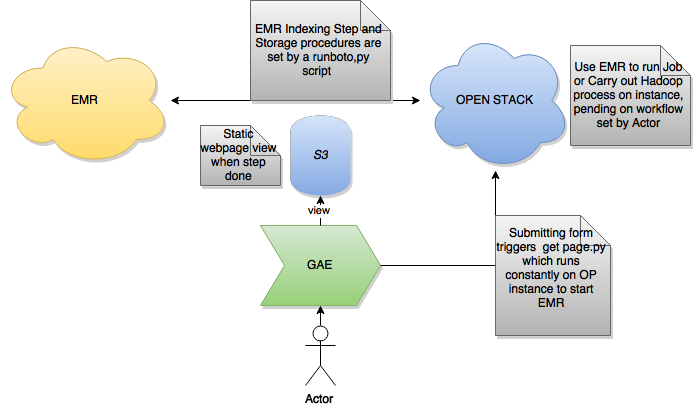
The results and log from the Open Stack Hadoop or EMR are generated to a text file that is uploaded to the S3. Either via aws CLI for Hadoop results and E.M.R. configured with boto on what folder for output and log.

The mapper’s job is to process the input data; the output is then passed to the reducer. The reducer processes the mapper’s output - creating new output values. Therefore stabilizing and grouping data using keys and values.

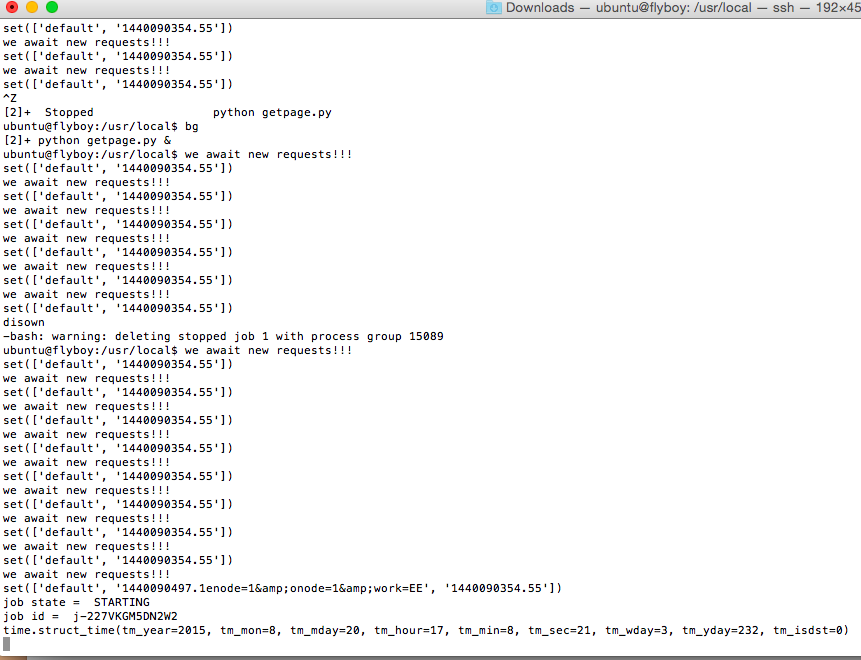
The Map-reduce groups its data by type, and splits into various nodes. It allows these nodes to work independently and then combined at once. These smaller chunks are normally based on the type of similar data found and then grouped together.

The algorithm used for generating index iterates through a list and only uses the position of the words in that window space. This produces only unique patterns and therefore saves memory by not including multiples of a key. These mapped values are printed with a ‘1’ and are incremented when reducing, and a repetition is found. The outcome, of a sort therefore means repeating values are identified at the bottom. During Matching, the key of the source documents would map the suspicious ones; so (abc times abc for example would produce abc square) and hence that is a suspicious document and this is identified since the key for the source is prevalent.

System Architecture is simple, with Script on Open Stack acting as a Controller for the System. Here conditions are checked and then script decides process to be carried out in the system flow.

**To Test**: Uploading values to the frontend then set controller on Open Stack Instance to keep running even when ssh terminal was closed. Fill in values on frontend and submit to see if EMR process has begun.

System Architecture is been designed so as to accommodate responsiveness therefore Front end would allow multiple requests been made while instances on open stack record changes made to front end and then add every request to array of requests awaiting to be launched on EMR.



# Scale Analysis

Whenever they are no requests made from the frontend, then no step is being added to the array of jobs waiting to be indexed. In a runboto.py script, it is programmed so whenever there are no steps (i.e. idleness) E.M.R. is terminated. This scales down the cost, by not making it run for longer and costing more, Indexing is handled in minutes and since pricing is hourly, it is better to run as many nodes in an hour than run say a 10 – hour cluster for 10 hours. Time is money for any institution, the faster – the better and other things can be done. It would cost just as much to run 100 clusters in 1 hour. Things can be polled to the hour that is busiest and utilized in that hour to save cost.

It takes at average 7 minutes to index 2 of the corpus suspicious documents. Using a main node (master) m1.large instance, 6 core m1.large instances, and 6 task m1.large instances (bid price at 0.004). The hadoop process splits the documents and results end up in 20 part files, each with a size of about 600kb.

Using this same setup as described above to index suspicious documents (39 of them), results in my bucket is same 20 part files but each with size of about 5.6mb. Surprising enough, this does not take far much time, as one would expect, EMR takes 8 minutes to handle this step. It is safe to say the system scales with the performance been not so different when compared to smaller files been indexed. Even the cost of storage and node costs where - Indexing these files does not cost any more storage as s3 ‘Data Transfer In’ is free.

Though, due to the process involved in such a system where indexed files may need to be download to Open Stack for matching. We are looking at a total size of 112mb (Suspicious Documents Index Result), and 1.8GB (301 Source Documents Index Result) in 13minutes. Where cost for downloading is $0.03 per GB. If such an amount of requests were run daily it would still be a quite cheap storage method for an Indexing system of this magnitude.

Altogether, total timing of Step 1 for Suspicious (8 minutes) and Step 2 for Source (13 minutes) that is 21 minutes are within the cost of E.M.R hour cost, assuming matching would be carried out in about the same time too, we are looking at still a cheap process but not so reliable if the user/students require immediate feedback. The logical assumption is throwing more machines (slave instances) would hasten the process and therefore cost more for the number of instances required.

So, I increased it to 10 core & 10 task instances for the whole Source-Documents and it indexes the 301 documents in 9 minutes. This is great for business as one can easily calculate and approximate for implementation of resources (time and money). It’s a no brainer that more machines would be quicker, so I tried 20 core & 20 task instances and my cluster was terminated with errors because the requested number of instances exceeds my EC2 quota!

Now I made use of the EMR\_EC2\_DefaultRole, which was a click away. Assuming, I needed to go faster i.e. my client requires it go faster, There would be a need to customize an EC2 for this purpose, which would possibly require implementing attachable volumes and enabling my instances for hadoop to find out how much it would cost my client’s business. If speed is paramount and client’s dollars is in pocket: 6core, 6task -13mins, 10core, 10 task – 9mins. Difference is in fours, so an assumption is EMR would need to be implemented to 18core, 18 tasks for a minute of processing. This I believe would be optimal for such a system.

Though installation of technology to carry out Hadoop jobs on Open Stack was successful but due to inadequate time available, experiments were not carried out on that very instance for Hadoop Processing. Only EMR was utilized, even though it hosted the controller to the whole system. It would have been interesting to see if EMR has been designed by AWS to be more efficient than a normal hadoop setup on any virtual machines. I assume it would produce similar results, though there are different factors that could affect/increase system efficiency, Number of machines would be the most crucial due to hadoop’s distributed file structure.

# Critque Approach

The final end product of the system comes short of some required features that were expected like producing an overview of detected text similarity for selected suspicious texts in comparison with the source texts (no match programs). Though station exists to be able to select 4 options for tasking like so (E-E, E-O, O-E or O-O) and specifying nodes to be used for each, are presently not functional.

It does implement all three of Google App Engine, EMR and Open Stack, which makes it possible to create new indexes while one is running. Thereby queuing several requests in array without impacting the responsiveness of front-end. This front end could have been more responsive and efficient in terms of interaction with the user and been able to present results back to the frontend for user to view, instead of visiting s3 link to find results.

Due to the God-factor (father time), even more important requirements like switching on a cloud resource (i.e. Open Stack Instance) to only when needed for indexing or matching where not top priority considering hassle in starting one back up with all necessary dependencies. Although this shouldn’t be a big deal as using a nova command to start one each time and using python commands such as os.system() could have been used to line up and install dependencies required each time. Better management of Open Stack might have been thoroughly considered if it wasn’t free.

Being able to specify window sizes, overlap and merge distance, number of nodes could have being possible by writing regex expressions to analyze a user post (example a post like: {time}enode=3&onode=4&work=EO&window=5&overlap=1) so as to pick variable and instead of using the same map reduce programs all the time. When a post is analyzed for values, ‘nano map.py’ a new index map program is assembled where, wherever window size is seen an edit of the new window size value replaces it. Program is saved and uploaded to s3 through cli/boto. Also, Merge distance value for the match program respectively. Same technique could have been used on ‘runboto.py’ program to specify the amount of node instances to be used for analysis.

Tedious processes as such were avoided due to little experience in server side programming, and most efforts were spent trying to investigate ways of handling this much complexity, in which included learning curve of python language, shell commands, boto syntax, and installing dependencies.

From the way posts are presently handled, this could be better if instead of fetching a webpage then adding it as a string to an array which is then used to boot EMR, a JSON file should be created every time a user fills the form in which explicitly describes a variable to its value in the object notation, this would make it easier to import this JSON file, such JSON file requests stored in S3 can then be used in the different programs that require it on the Open Stack Server.

Present Application can be improved by persistent storage from the front end. GAE can be connected to the Google App Cloud, where a database of logs would be generated from the number of requests made. This would help build a more stable application where by different entities talk to each other directly. Another option I prefer would be building the whole application on open Stack which is more of a private cloud, still use boto to talk to AWS as this is efficient enough and a private cloud making calls to AWS which can be more secure and cheaper if there’s more traffic than anticipated.

The system would allow things to be more seamless if

When a user requests a workflow of say EE, two steps should be generated automatically to occur sequentially to handle the Indexing & Matching.

Securities in cloud services do not really have much basis for now but in the nearest future, new extensions should be easily implemented and connected to the system, as we know security could be a prominent issue.

Implementing Spot instances on the fly as processing goes on would have made the system more scalable and efficient. Meaning the master node should be able to add more slave nodes to support the system whenever processing starts to take more time than expected. Also cut slave nodes when costing past limit than originally planned with client. Functions like that would really bring the management of servers to their full potential.

##### References

1. 2. Schneier B (2009) Cloud computing. Accessed 11th February, 2014 from http://www.schneier.com/blog/archives/2009/06/cloud\_computing.html
2. https://hadoop.apache.org/docs/r1.2.1/hdfs\_design.html